Separation of Privilege

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-12-06 L4 / D/P, L[5]

A system should ensure that multiple conditions are met before granting permissions to an object. Checking access on only one condition may not be adequate for strong security. If an attacker is able to obtain one privilege but not a second, he or she may not be able to launch a successful attack. If a software system largely consists of one component, the idea of having multiple checks to access different components cannot be implemented. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" on page 9:

> Separation of privilege: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extendible protected data types usually depend on separation of privilege for their implementation.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in the "Principle of Separation of Privilege" section on pages 347-348:[9]

> This principle is restrictive because it limits access to system entities.
>
> Definition 13-6, The principle of separation of privilege states that a system should not grant permission based upon a single condition.
>
> This principle is equivalent to the separation of duty principle discussed in Section 6.1 [of *Computer Security*]. Company checks for over $75,000 must be signed by two officers of the company. If either does not sign, the check is not valid. The two conditions are the signatures of both officers.
>
> Similarly, systems and programs granting access to resources should do so when more than one condition is met. This provides a fine grained control over the resource, and additional assurance that the access is authorized.
>
> **Example 1.** On Berkeley-based versions of the UNIX operating system, users are not allowed to change from their account to the root account unless two conditions are met. The first is that the user knows the root password. The second is that the user is in the wheel group (the group with GID 0). Meeting either condition is not sufficient to acquire root access. Meeting both conditions is required.

According to NIST [NIST 01] in Section 3.3, "IT Security Principles," on page 12:

> **Limit or Contain Vulnerabilities.**

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

> Design systems to limit or contain vulnerabilities. If a vulnerability does exist, damage can be limited or contained, allowing other information system elements to function properly. Limiting and containing insecurities also helps to focus response and reconstitution efforts to information system areas most in need.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in the "Principle 5: Compartmentalize" section on pages 102-104:[10]

> The basic idea behind compartmentalization is to minimize the amount of damage that can be done to a system by breaking the system into as few units as possible while still isolating code that has security privileges. This same principle explains why submarines are built with many different chambers, each separately sealed. If a breach in the hull causes one chamber to fill with water, the other chambers will not be affected. The rest of the ship can keep its integrity, and people can survive by making their way to parts of the submarine that are not flooded. Unfortunately, this design doesn't always work, as the Kursk disaster shows.
>
> Another common example of the compartmentalization principle shows up in prison design. Prison designers try hard to minimize the ability for large groups of convicted criminals to get together. Prisoners don't bunk in barracks, they bunk in cells of up to two. Even when they do congregate, say in a mess hall, that's the time and place where other security measures are increased to help make up for the large rise in risk.
>
> In the computer world, it's a lot easier to point out examples of poor compartmentalization than it is to find good examples. The classic example of how not to do it is the standard UNIX privilege model, where interesting operations work on an "all or nothing" basis. If you have root privileges, you can do anything you want anywhere on the system that you want. If you don't have root access, there are significant restrictions. As we mentioned, you can't bind to ports under 1024 without root access. Similarly, you can't access many operating system resources directly (for example, you have to go through a device driver to write to a disk, you can't deal with it directly).
>
> Given a device driver, if an attacker exploits a buffer overflow in the code, the attacker can make raw writes to disk, and mess with any data in the kernel's memory. There are no protection mechanisms to prevent that. Therefore, it is not possible to support a log file on a local hard disk that can never be erased, so that you can keep accurate audit information up until the time of a break-in. Attackers will always be able to circumvent any driver you install, no matter how well it mediates access to the underlying device.
>
> On most platforms, it is not possible to protect one part of the operating system from others. If one part is compromised, then everything is hosed. Very few operating systems do compartmentalize. Trusted Solaris is a well-known one, but it is unwieldy. In those operating systems with compartmentalization, operating system functionality is broken up into a set of roles. Roles map to entities in the system that need to provide particular functionality. One role might be a LogWriter role, which would map to any client that needs to save secure logs. This role would be associated with a set of privileges. For example, a LogWriter may have permission to append to its own log files, but never to erase from any log file. Perhaps only a special utility program will be given access to the LogManager role, which would have complete access over all the logs. Standard programs would not have access to this role. Even if an attacker breaks a program and ends up in the operating system, the attacker still won't be able to mess with the log files unless the log management program gets broken too.
>
> Complicated "trusted" operating systems are not all that common. One reason is that this kind of functionality is difficult to implement and hard to manage. Problems like dealing with memory protection inside the operating system provide challenges that have solutions, but not ones that are simple to effect.

10.

The compartmentalization principle must be used in moderation. If you segregate each little bit of functionality, then your system will become completely unmanageable.

"What Goes Wrong"

According to McGraw and Viega [McGraw 03c]:[13]

> Changing the root directory in Unix processes doesn't work unless you follow up.
>
> The chroot() system call provides a standard way to compartmentalize Unix processes. This call changes the root directory for all subsequent file operations, establishing a "virtual" root directory. Unfortunately, chroot() usually doesn't work well in practice.
>
> One flaw is that only root can use it. Often, programmers will allow the program to continue to run without totally dropping root privileges—a bad idea. When you run a process chroot(), the process should immediately set both the effective user ID (EUID) and UID to a less-privileged user to eliminate the window of vulnerability. Also, chroot() does not work exactly as advertised unless you immediately follow it with a call to chdir("/"). Without that, an attacker may be able to use relative paths to access the rest of the file system.
>
> Obviously, chroot can be a maintenance nightmare. Make sure you avoid putting any setuid root program in a chroot environment. When such programs are available in a jail, the jail is likely to be broken from the inside out.

Separation of privilege is defined differently by Howard and LeBlanc [Howard 02]. We include their definition to show the importance of having multiple processes working together with different levels of privileges. This excerpt is from Chapter 3, "Security Principles to Live By," in the "Separation of Privilege" section on pages 61-62:[14]

> An issue related to using least privilege is support for separation of privilege. This means removing high privilege operations to another process and running that process with the higher privileges required to perform its tasks. Day-to-day interfaces are executed in a lower privileged process.
>
> In June 2002, a severe exploit in OpenSSH v2.3.1 and v3.3, which ships with versions of Apple Mac OS X, FreeBSD and OpenBSD, was mitigated in v3.3 because it supports separation of privilege by default. The code that contained the vulnerability ran with lower capabilities because the UsePrlvilegeSeparation option was set in sshd_config. You can read about the issue at http://www.openssh.com/txt/preauth.adv.
>
> Another example or privilege separation is Microsoft Internet Information Services (lIS) 6, which ships in Windows .NET Server. Unlike lIS 5, it does not execute user code in elevated privileges by default. All user mode HTTP requests are handled by external worker processes (named w3wp.exe) that run under the Network Service account, not under the more privileged Local System account. However, the administration and process management process, inetinfo.exe, which has no direct interface to HTTP requests, runs as Local System.
>
> The Apache Web Server is another example. When it starts up, it starts the main Web server process, httpd, as root and then spawns new httpd processes that run as the low privilege nobody account to handle the Web requests.

References

[Bishop 03] Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.

13. All rights reserved. It is reprinted with permission from CMP Media LLC.
14. From *Writing Secure Code, Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.

[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, Second Edition*. Redmond, WA: Microsoft Press, 2002.
[McGraw 03c]	McGraw, Gary & Viega, John. "Divide and Conquer." *Software Development.* CMP Media LLC, April, 2003.
[NIST 01]	NIST. *Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com